An Optimal Parallel Algorithm for Integer Sorting

John H. Reif†
Aiken Computation Lab.
Harvard University
Cambridge, Massachusetts

0. ABSTRACT

We assume a parallel RAM model which allows both concurrent writes and concurrent reads of global memory. Our algorithms are *randomized:* each processor is allowed an independent random number generator. However our stated resource bounds hold for worst case input with overwhelming likelihood as the input size grows.

We give a new parallel algorithm for integer sorting where the integer keys are restricted to at most polynomial magnitude. Our algorithm costs only logarithmic time and is the first known where the product of the time and processor bounds are bounded by a linear function of the input size. These simultaneous resource bounds are asymptotically optimal. All previous known parallel sorting algorithms required at least a linear number of processors to achieve logarithmic time bounds, and hence were nonoptimal by at least a logarithmic factor.

1. INTRODUCTION

1.1. Sequential Sorting Algorithms

The literature on sequential algorithms generally assumes the sequential Random Access Machine (RAM); see [Aho, Hopcroft, and Ullman, 74]. In a single step a RAM may either read or write into a memory cell or register, or perform an arithmetic operation on an integer. Each memory cell and register may contain a binary number with at most a logarithmic number of bits in the input size, (this restriction is found in practice since normally the machine word size is made only large enough to hold a constant number of addresses to the memory) and hence the stored integers can have at most polynomial magnitude.

A large literature exists on efficient sequential RAM algorithms with time bound linear in the input size. Many of these algorithms require sorts to be done on integers of at most polynomial magnitude. For example, the depth first search algorithms of [Tarjan, 72] and [Hopcroft and Tarjan, 73] require the edges (which may be considered integers) to be sorted into adjacency lists. A $\Omega(n \log n)$ comparison sort such as QUICK-SORT or HEAP-SORT would not be sufficiently efficient for these applications. Instead, the BUCKET-SORT (see [Aho, Hopcroft, and Ullman, 74]) is used to sort in linear time. The BUCKET-SORT algorithm is sufficiently simple and elegant so that it is widely used in practice.

The goal of this paper is to develop an efficient and possibly practical integer sorting algorithm for a parallel RAM model, but we will utilize quite different techniques--such as randomization.

1.2. Known Parallel Sorting Algorithms

The performance of a parallel algorithm can be specified by bounds on its principal resources: processors and time. We generally let P denote the processor bound and T denote the time bound. The product PT is lower bounded by at least a constant times the minimum sequential time S required to solve the problem. Thus we consider a parallel algorithm to be (asymptotically) *optimal* if $PT = O(S)$. Of course, if we have an optimal algorithm with any processor bound P, then we also have (by the obvious processor simulation) an optimal algorithm for any processor bound less than P. Hence an optimal algorithm may also be useful in practical situations where we have a limited number of processors.

We assume a *parallel RAM* model of [Shiloach and Vishkin, 81]. The processors are synchronous, and each is a unit cost sequential RAM as defined above.

† This work was supported by Office of Naval Research Contract N00014-80-C-0647 and National Science Foundation grant DCR-8503251.

0272-5428/85/0000/0496$01.00 © 1985 IEEE

This parallel RAM model allows multiple reads at a single memory cell and also allows multiple writes at a single memory cell, where multiple writes are allowed to be resolved arbitrarily. This model is known as the CRCW parallel RAM and is quite robust, see [Kucera, 82] for its relation to other parallel machine models. In addition we allow the parallel RAM to be *randomized* by equipping each processor with an independent random number generator. (See [Reif, 84a], [Reif, 84b] for a description of randomized parallel computations, and see [Reif and Tyger, 85] for a method for efficient parallel generation of pseudo random numbers.)

There are a number of known algorithms for sorting in logarithmic time using a linear number of processors; (1) [Reischuk, 82] gave a randomized parallel RAM algorithm which unfortunately requires memory cells of $n^{1/2}$ bits each rather than the $O(\log n)$ bits usually assumed, (2) [Reif and Valiant, 83] gave a randomized parallel algorithm FLASH-SORT which has only moderate constant bounds and requires memory cells of $O(\log n)$ bits each, and (3) [Ajtai, Komlós, and Szemerédi, 83; and Leighton, 84] give a parallel algorithm AKS which has some good theoretical properties such as being deterministic, but has impractically large constant factors. In theory, these results appeared to finally settle the problem of parallel sorting since $PT = \Omega(n \log n)$ is a known lower bound in the case of comparison sorting. However, these lower bounds on PT need not hold for *integer sorting:* sorting n integers on the range $[n^{O(1)}]$.† Note that the restriction to the range $[n^{O(1)}]$ is natural, since RAM memory cells can only contain numbers of polynomial magnitude. The lower bounds $PT = \Omega(n)$ for integer sorting follow from the fact that each input key costs at least one unit operation on some processor. As in the sequential case, integer sorting is all that is required for many practical parallel applications of interest.

1.3. Our Optimal Parallel Sorting Algorithm

Our main result is an optimal randomized parallel algorithm for integer sorting requiring $\tilde{O}(\log n)$ time, and $n/\log n$ processors. Here $\tilde{O}$ denotes that the upper bound holds within a constant factor with overwhelming likelihood, for the *worst case input.* In particular, we let $T(n) = \tilde{O}(f(n))$ denote there exists c for all $\alpha \geq 1$, for all sufficiently large n, $T(n) \leq c\alpha f(n)$ holds with probability at least $1 - 1/n^{\alpha}$.

† Note throughout this paper, we let $[n]$ denote $\{1,...,n\}$.

Our integer sorting algorithm is quite easy to implement and may be of some practical use, since it has very moderate constant factors. It achieves its efficiency by some interesting new randomization techniques.

1.4. Applications

As an almost immediate consequence of integer sorting algorithm we derive an optimal $O(\log n)$ time, $n/\log n$ processor algorithm for computing a random permutation, which previously was an open problem [Vishkin, 84].

1.5. Sublogarithmic Algorithms

Although our bound $PT = O(n)$ for $T \geq O(\log n)$ is optimal for integer sorting, this linear bound does not seem to be achieveable for sublogarithmic time bounds. We give an algorithm for prefix sum computation which takes time $O((\log n)/\text{loglog}(P \log n/n))$ using $P \geq n/\log n$ processors. Thus for any $\epsilon > 0$ in the case we have $P = n$ processors, our time bound is $T = (\log n/\text{logloglog } n)$, and in the case $P = n^{1+\epsilon}$, our time bound is $T = O(\log n/\text{loglog } n)$. Our parallel algorithm for integer sort utilizes prefix computation in its crucial computations and can straightforwardly be modified by similar methods to achieve these sublogarithmic time bounds.

1.6. Organization of This Paper

In Section 2, we discuss known optimal results for parallel prefix computation which will be of some use in devising our optimal parallel algorithms. In Section 3, we give our optimal parallel algorithm for integer sorting. In Section 4, we give an application to finding a random permutation. In Section 5, we describe some sublogarithmic algorithms. In the Appendix A we give some useful upper bounds for the tails of various probability distributions which arise in the analysis of our randomized algorithms.

2. PARALLEL PREFIX COMPUTATION

2.1. Prefix Circuits

Let Σ be a domain and let $\cdot$ be an associative operation which takes $O(1)$ sequential time over this domain. The *prefix computation problem* is defined as follows:

input $(X(1),...,X(n)) \in \Sigma^n$

output $(X(1),X(1)\cdot X(2),...,X(1)\cdot\cdot\cdot X(n))$.

[Ladner and Fischer, 80] show prefix computation can be done by a circuit of size n and depth $O(\log n)$. The following is a straightforward generalization (see [Fich, 82]).

PROPOSITION 2.1. *For each $n \geq 1$ and $2 \leq b \leq n$, there exists a circuit $\mathbf{C}_{n,b}$ for prefix computation with fan-in b, with $O(n)$ nodes and $O((\log n)/\log b)$ depth.*

2.2. Optimal Prefix Computation

Known techniques attributed to Brent, give the following processor improvement:

LEMMA 2.1. *Prefix computation can be done in time $O(\log n)$ using $n/\log n$ P-RAM processors.*

Proof. Without loss of generality we can assume n is a power of two. Let the set of processors be $\Pi = \{1,...,P\}$ where $P = \lceil n/\log n \rceil$. Let each processor $\pi \in \Pi$ compute $Y(\pi,1) = X(1+(\pi-1)\log n)$, and $Y(\pi,i) = Y(\pi,i-1)X(i+(\pi-1)\log n)$ for $i = 1,...,\log n$ where $i + (\pi-1)\log \leq n$. We can easily simulate the prefix circuit $\mathbf{C}_{P-1,2}$ with inputs $Y(1,\log n),...,Y(P-1,\log n)$ to compute $Z(1),...,Z(P-1)$ where $Z(j) = Y(1,\log n)\cdot\cdot\cdot Y(j,\log n)$ for $j = 1,...,P-1$. This circuit simulation takes at most time $O(\log n)$ using P processors. Finally, output $(Y(1,1),...,Y(1,\log n),$ $Z(1)\cdot Y(2,1),...,Z(1)\cdot Y(2,\log n),...,$ $Z(P-1)\cdot Y(P,1),...,Z(P-1)\cdot Y(P,\log n))$ $= (X(1),X(1)\cdot X(2),...,X(1)\cdots X(n))$. □

The *prefix sum computation problem* is defined as follows: Given input integers $X(1),...,X(n) \in [n]$, output the vector $\text{PREFIX-SUM}(X) = (Y(0),Y(1),...,Y(n))$ where $Y(0) = 0$ and $Y(i) = \sum_{j\leq i} X(j)$ for $i \in [n]$. By Lemma 2.1, we can do this computation in time $O(\log n)$ using $n/\log n$ processors.

3. AN OPTIMAL PARALLEL SORTING ALGORITHM

3.1. Known Sorting Algorithms

The *integer sorting problem* of size n is defined:

input keys $k_1,...,k_n \in [n^{O(1)}]$

output permutation $\sigma = (\sigma(1),...,\sigma(n))$ such that $k_{\sigma(1)} \leq \cdot\cdot\cdot \leq k_{\sigma(n)}$.

The input keys $k_1,...,k_n$ are not necessarily distinct. By use of the well known and quite practical BUCKET-SORT algorithm [Aho, Hopcroft, and Ullman, 74],

LEMMA 3.1. *Integer sorting can be done in time $O(n)$ by a deterministic sequential RAM.*

Any comparison based sort requires $PT = \Omega(n \log n)$, and the most efficient parallel sorts actually achieve these bounds. In particular, [Reif and Valiant, 83] show

LEMMA 3.2. *n keys can be sorted in time $\tilde{O}(\log n)$ using n processors.*

This algorithm uses memory cells of $O(\log n)$ bits. It can be implemented on a constant degree network as well as by the randomized P-RAM model (this algorithm can be considerably simplified if implemented by a P-RAM). In addition, [Ajtai, Komlós, and Szemerédi, 83], [Leighton, 84] give a deterministic sorting network which takes $O(\log n)$ time with $O(n)$ processors. These sorts are comparison based and don't use the fact that the input keys $k_1,...,k_n$ are integers in $[n^{O(1)}]$ as in the case of most applications. In the following, we prove:

THEOREM 3.1. *Integer sort can be done in time $\tilde{O}(\log n)$ using $n/\log n$ P-RAM processors.*

We will achieve $PT = \tilde{O}(n)$ for integer sorting, making essential use of randomization. We would be quite surprised if any purely deterministic methods yield $PT = O(n)$ for parallel integer sort in the case of time bounds $T = O(\log n)$. Although we will use deterministic methods to solve some restricted integer sorting problems, (see Lemmas 3.4 and 3.6 below) our optimal parallel algorithm for the general integer sorting problem requires some interesting, new use of randomization techniques (see Lemmas 3.8 and 3.9).

3.2. Small Integer Sorting Problems

A sorting algorithm is *stable* if given $k_1,...,k_n$, the algorithm outputs a permutation σ of $(1,...,n)$ where for all $i, j \in [n]$ if $k_i = k_j$ and $i < j$ then $\sigma(i) < \sigma(j)$.

Given a sequence of keys $k_1,...,k_n \in [n]$ let the *key index set* for key value $k \in [n]$ be $I(k) = \{i | k_i = k\}$. We will assume W.L.G. $\log n$ divides n.

LEMMA 3.3. *Given $I(1),...,I(n)$, we can sort $k_1,...,k_n$ in $O(\log n)$ time using $P = n/\log n$ processors.*

Proof. Compute $(h_0,...,h_k) = \text{PREFIX–SUM}(|I(1)|, \ldots, |I(n)|)$ in $O(\log n)$ time using P processors by Lemma 2.1. We then set $\sigma(h_{k-1}+1),...,\sigma(h_k)$ to consecutive elements in $I(k)$ using a total of $O(\log n)$ time and P processors (the required processor assignment can easily be done by using the prefix sum computation by Lemma 2.1.) Then $k_{\sigma(1)} \leq ... \leq k_{\sigma(n)}$ so σ gives a sort. □

LEMMA 3.4. *A stable sort of n keys $k_1,...,k_n \in [\log n]$ can be computed in $O(\log n)$ time using $P = n/\log n$ processors.*

Proof. To each processor $\pi \in [P]$, we assign key indices $J(\pi) = \{j | (\pi-1)\log n < j \leq \min(n, \pi \log n)\}$. Let each processor π sequentially sort the keys $\{k_j | j \in J(\pi)\}$ by BUCKET-SORT in time $O(\log n)$, and so compute each list $J_{\pi,k} = (j \in J(\pi) | k_j = k)$ in increasing order of indices for each key value $k \in [\log n]$. Then for each key value $k \in [\log n]$ we compose the lists $J_{1,k}...J_{P,k}$ to form the list $I(k)$ of indices with key value k. Finally, we apply Lemma 3.3 to compute the required permutation σ ordering the indices as they appear in $I(1),...,I(P)$. The total time is $O(\log n)$ using P processors. □

LEMMA 3.5. *If n keys in $[D]$ can be stably sorted in $O(\log n)$ time using $P = n/\log n$ processors, then n keys $k_1,...,k_n \in [D^2]$ can be sorted in $O(\log n)$ time using the same number of processors.*

Proof. Let $k_i' = \lceil k_i/D \rceil + 1$ and let $k_i'' = k_i - (k_i'-1)D+1$ for each $i \in [P]$. We first sort $k_1',...,k_n'$, yielding a permutation σ. Then we sort $k''_{\sigma(1)}, \ldots, k''_{\sigma(n)}$, yielding a permutation σ'. Then $k_{\sigma'(1)} \leq ... \leq k_{\sigma'(n)}$, and hence σ' is a sort of $k_1,...,k_n$. □

This immediately implies:

LEMMA 3.6. *n keys $k_1,...,k_n \in [(\log n)^2]$ can be sorted in $O(\log n)$ time using $P = n/\log n$ processors.*

LEMMA 3.7. *The integer sorting problem of size n for key values in $[n^c]$ can be solved by c consecutive stable integer sorts of size n for key values in $[n]$.*

3.3. Randomized Sampling and Sorting in Key Domain $[n/(\log n)^2]$

In the following subsection, we fix a key domain $[D]$ where $D = n/(\log n)^2$. (We assume W.L.G. $(\log n)^2$ divides n). Let the input keys be $k_1,...,k_n \in [D]$ and their index sets be $I(k) = \{i | k_i = k\}$ for each key value $k \in [D]$.

LEMMA 3.8. *Given as input $k_1,...,k_n \in [D]$, we can compute $N(1),...,N(D)$ in $\tilde{O}(\log n)$ time using $P = n/\log n$ processors, such that $\sum_{k \in [D]} N(k) \leq O(n)$ and furthermore with high likelihood (in fact with probability $\geq 1-1/n^\alpha$ for any given $\alpha \geq 1$) $N(k) \geq |I(k)|$ for each $k \in [D]$.*

As proof, we execute the following randomized sampling algorithm

Step 1 for each processor $\pi \in [P]$ in parallel

do choose a random $s_\pi \in [n]$ od

$S \leftarrow \{s_1,...,s_P\}$

Comment. Here we randomly choose a set $S \subseteq [n]$ of P key indices.

Step 2 Sort $k_{s_1},...,k_{s_P}$ and compute index set $I_S(k) = \{i \in S | k_i = k\}$ for each key value $k \in [D]$.

Comment. Applying Lemma 3.2, this sorting can be done by known parallel algorithms in $\tilde{O}(\log n)$ time using P processors.

Step 3 for each $k \in [D]$ do

$N(k) \leftarrow d_0(\log n)\max(|I_S(k)|, \log n)$ od

Comment. d_0 is a constant to be determined below in the probabilistic analysis.

output $N(1),...,N(D)$.

Next we prove the probabilistic bounds given in Lemma 3.8.

Proof. If $d_0(\log n)^2 \geq |I(k)|$ then always $N(k) \geq d_0(\log n)^2 \geq |I(k)|$. Else suppose $d_0(\log n)^2 < |I(k)|$. $|I_S(k)|$ is upper bounded by a binomial variable with parameters $n/\log n$, $|I(k)|/n$. The Chernoff bounds given in Appendix A, Lemma A.1, imply there exists c for all $\alpha \geq 1$, $\text{Prob}(|I_S(k)| \geq c\alpha|I(k)|/\log n) \geq 1-1/n^\alpha$. Setting $d_0 = (c\alpha)^{-1}$, since $N(k) \geq d_0|I_S(k)|\log n$, the probability bound $\text{Prob}(N(k) \geq |I(k)|) \geq 1 - 1/n^\alpha$ holds as claimed. □

Lemma 3.9. *n keys $k_1,...,k_n \in [D]$, where $D = n/(\log n)^2$ can be sorted in $O(\log n)$ time using $P = n/\log n$ processors.*

Proof. (We will actually use $O(P)$ processors, but we observe that we can then slow the computation down by a constant factor to reduce the processor bound to P.) Our randomized algorithm is given below.

Step 1 Compute $N(1),...,N(D)$ as defined in Lemma 3.8.

Comment. Here we use the random sampling algorithm.

Step 2

$(\overline{N}(0),...,\overline{N}(D)) \leftarrow$ PREFIX–SUM$(N(1),...,N(D))$

Comment. This prefix-sum computation is done by Lemma 2.1 in $O(\log n)$ time and $O(P)$ processors.

Step 3 for each key value $k \in [D]$ do

$(\overline{N}(0),...,\overline{N}(D)) \leftarrow$ PREFIX–SUM$(N(1),...N(D))$.

Comment. This prefix-sum computation is done by Lemma 2.1 in $O(\log n)$ time and $O(P)$ processors.

Step 3 for each key value $k \in [D]$ do

$P_k \leftarrow \{\pi | \pi \in [D]$ or $\overline{N}(k-1)+D < \pi \leq \overline{N}(k)+D\}$.

Using these P_k processors, construct a table $A_k = (A_k(1),A_k(2),...,A_k(N(k)), A_k(N(k)+1))$ and initialize each element of the table to be an empty list.

od

Step 4 for each $\pi \in [P]$ in parallel do

for each $t = 1,...,\log n$ sequentially do

$i_\pi \leftarrow (\pi-1)\log n+t$

choose a random number $r_\pi \in [N(k)]$

attempt to add i_π to front of list $A_{k_{i_\pi}}(r_\pi)$

if successful (i.e., i_π is now in the front of list $A_{k_{i_\pi}}(r_\pi)$)

then CONFLICT$(i_\pi) \leftarrow 0$

else CONFLICT$(i_\pi) \leftarrow 1$ fi

od od

Comment. Each processor $\pi \in [P]$ is responsible for keys $k_{(\pi-1)\log n+1},...,k_{\pi \log n}$. The inner loop for $t = 1,...,\log n$ is executed sequentially so as to minimize conflicts. In the t-th iteration of the inner loop, processor π attempts to add the index $i_\pi = (\pi-1)\log n+t$ of the key k_{i_π} to the front of list $A_{k_{i_\pi}}(r_\pi)$ where r_π is a randomly chosen integer in $[N(k)]$. This may not be successful if some other processor π' simultaneously attempts to add some other index $i_{\pi'}$ to the front of list $A_{k_{i_\pi}}(r_\pi)$. Only one addition to this list will succeed. But this conflict will only happen in the case $k_{i_{\pi'}} = k_{i_\pi}$ and π' makes the same unlucky choice of $r_{\pi'} = r_\pi$.

Claim 3.1. Let $n' = \sum_{i=1}^{n} \text{CONFLICT}(i)$. Then $n' \leq \tilde{O}(P)$. In particular, for all c for all $\alpha \geq 1$ $\text{Prob}(n' \leq \alpha\, cn/\log n) \geq 1-1/n^\alpha$.

Proof. By Lemma 3.8, with likelihood $\geq 1-1/n^\alpha$, we can assume $N(k) \geq |I(k)|$. Let $n_k = \sum_{i \in N(k)} \text{CONFLICT}(i)$. The key observation is that on each stage, $1/\log n$ of the key indices of $I(k)$ are assigned to random positions of the table A_k. Let $n_{k,t}$ be the number of indices $i \in N(k)$ where CONFLICT(i) is set to 1 on stage t. Then by definition $n_k = \sum_{t=1}^{\log n} n_{k,t}$. We now apply the probabilistic bounds given in Appendix A, and we consider upper bounds on probability variables to be over the range of probability densities from $1/n^\alpha$ to $1-1/n^\alpha$. By Lemma A.3, each $n_{k,t}$ is upper bounded by a hypergeometric variable with parameters $|I(k)|/\log n$, $|I(k)|/\log n$, $|I(k)|$. Then Lemma A.4 implies each $n_{k,t}$ is upper bounded by a binomial variable with parameters $N(k)/\log n$, $1/\log n$. Hence by (Hoeffding's inequality) Lemma A.2, $n_k = \sum_{t=1}^{\log n} n_{k,t}$ is upper bounded by a binomial variable with parameters $N(k)$, $1/\log n$. Furthermore $\sum_{k \in [D]} N(k) \leq O(n)$, so $\sum_{k \in [D]} n_k$ is upper bounded (again by Hoeffding's inequality) by a binomial variable with parameters $O(n)$, $1/\log n$. The Chernoff bounds given in Lemma A.1 immediately imply the claimed probabilistic bounds on n'. □

We now continue the algorithm used to prove Lemma 3.9.

Step 5 $(u(0),...,u(n)) \leftarrow$
PREFIX–SUM(CONFLICT(1),..., CONFLICT(n))
$n' \leftarrow u(n)$
for each $\pi \in [P]$ in parallel do
for each $t = 1,...,\log n$ sequentially do
$i_\pi \leftarrow (\pi-1)\log n+t$
if CONFLICT(i_π) then $j_{u(i_\pi)} \leftarrow i_\pi$ fi
od od

Comment. $(j_1,...,j_{n'})$ is the list of indices j such that CONFLICT(j) = 1. Again, the prefix computations can be done by applying Lemma 2.1.

Step 6 Sort $k_{j_1},...,k_{j_{n'}}$ and for each key value $k \in [D]$ assign $A_k(N(k)+1)) \leftarrow \{j_m | k=k_{j_m}\}$.

Comment. In $A_k(N(k)+1)$ we place the list $\{j_m | k=k_{j_m}\}$ of conflicted indices with key value k. Assuming $n' \leq O(P)$, this step can be done by Lemma 3.2 using known parallel sorting algorithms in time $\tilde{O}(\log n)$ using P processors.

Step 7 for each key value $k \in [D]$ do
Construct table A_k' consisting of a list of all the elements of the lists
$A_k(1),A_k(2),...,A_k(N(k)),A_k(N(k)+1)$.
od

Comment. This is done in $O(\log n)$ time by careful use of the processor set P_k. In particular, we first compute
$(a_k(0),...,a_k(N(k)+1)) \leftarrow$ PREFIX–SUM($|A_k(1)|$, $|A_k(2)|, \ldots, |A_k(k)|, |A_k(N(k)+1)|$). Note that $|A_k(i)| \leq d_0 \log n$ for each i. Hence for each $i = 1,...,N(k)+1$ in parallel we can place the elements of $A_k(i)$ into locations $A_k'(a_k(i-1)+1),....,A_k'(a_k(i))$ using a single processor $\pi \in P_k$ with time $O(\log n)$.

Step 8 Compute a permutation σ of $(1,...,n)$ such that the elements of $A_1',...,A_D'$ appear in order.

Comment. We apply here Lemma 3.3.

output $\sigma = (\sigma(1), \ldots, \sigma(n))$.

The total time for steps 1-8 is $\tilde{O}(\log n)$ using P processors. □

3.4. Summary of Our Parallel Sorting Algorithm

Finally, we prove Theorem 3.1, by combining the above techniques. We again assume W.L.G. $(\log n)^2$ divides n, and by Lemma 3.7 we can assume W.L.G. the key values are in $[n]$.

Input keys $k_1,...,k_n \in [n]$

Step 1 Assign $k_i' = \lceil k_i/(\log n)^2 \rceil + 1$ and $k_i'' = k_i - (k_i'-1)(\log n)^2 + 1$ for each $i \in [n]$

Comment. $k_1',...,k_n' \in [D]$ where $D = n/(\log n)^2$ and $k_1'',...,k_n'' \in [(\log n)^2]$

Step 2 Sort $k_1',...,k_n' \in [D]$ resulting in index sets $I(k) = \{i | k_i' = k\}$ for each key value $k \in [D]$

Comment. This is done by applying Lemma 3.9.

Step 3 Sort $\{k_i'' | i \in I(k)\} \subseteq [(\log n)^2]$ yielding ordered list $L(k)$ of indices in $I(k)$ for each key value $k \in [D]$

Comment. This is done by applying the stable sort of Lemma 3.6 to the ordered list of keys $I(1)...I(D)$.

Step 4 Compute the permutation σ which orders the indices as $L(1),...,L(D)$

Comment. Here we apply Lemma 3.3, yielding σ satisfying $k_{\sigma(1)} \leq \ldots \leq k_{\sigma(n)}$

output σ

The Lemmas 3.2-3.9 and the appropriate use of prefix-sum computation (Lemma 2.1) imply that each step can be done in $\tilde{O}(\log n)$ using $P = n/\log n$ processors. □

4. OPTIMAL PARALLEL GENERATION OF A RANDOM PERMUTATION

COROLLARY 4.1. *A random permutation σ of $(1,...,n)$ can be constructed in $\tilde{O}(\log n)$ time using $P = n/\log n$ P-RAM processors.*

Proof. We execute the following algorithm.

Step 1 for each processor $\pi \in [P]$ in parallel do
for each $t = 1,...,\log n$ do
$i_\pi \leftarrow (\pi-1)\log n+t$
randomly choose $k_{i_\pi} \in [P]$
od od

Step 2 Sort $k_1,...,k_n$ and compute $I(k) = \{i | k_i = k\}$ for each key value $k \in [P]$

Comment. The sort can be done by Lemma 3.1 in time $\tilde{O}(\log n)$ using P processors.

Claim 3.2. With high likelihood, $|I(k)| \leq O(\log n)$ for each $k \in [P]$. In particular, there exists c for all $\alpha \geq 1$ Prob($|I(k)| \leq c\alpha \log n) \geq 1-1/n^\alpha$.

Proof. Each $|I(k)|$ is upper bounded by a binomial variable with parameters n, $\log n/n$. Hence the claimed bounds follow from the Chernoff bounds of Lemma A.1. □

Step 3 for each $\pi \in [P]$ in parallel do

let $L(k)$ be a random permutation of the elements of $I(k)$

od

Comment. A random permutation $I(k)$ can easily be sequentially computed in $O(|I(k)|)$ time by a single processor.

Step 4 Compute $\sigma = (\sigma(1),...,\sigma(n))$, the permutation of $(1,...,n)$ which gives the order of appearance of the indices in $L(1),...,L(P)$.

Comment. This can be done in $O(\log n)$ time by Lemma 3.3.

output random permutation σ

The total time for the steps 1-4 is $\tilde{O}(\log n)$ using P processors. □

5. SUBLOGARITHMIC TIME ALGORITHMS

Lemma 2.1 described an optimal $O(\log n)$ time, $n/\log n$ processor algorithm for prefix computation. Now we show that the computation can be done in sublogarithmic time using slightly more processors.

THEOREM 5.1. *Prefix Sum Computation can be done in time* $O((\log n)/\text{loglog}(P\log n/n))$ *using* $P \geq n/\log n$ *P-RAM processors with many cells containing* $O(\log P)$ *bits each.*

Note if we limit the memory cells to $O(\log n)$ bits each, then we must limit processors to $P \leq n^{O(1)}$.

Proof. Suppose we are input integers $X(1),...,X(n) \in \{1,...,n\}$. Fix $s = \text{loglog}((P \log n)/n)$, $d = \lceil \log n/s \rceil$, $n' = n/d$, and $b = \lceil \log(P/n')/(s+2) \rceil$. We can assume W.L.G. that n' is an integer power of b (else the length of the input need be increased by at most a constant factor). In time $O(d)$ using $n' \leq P$ processors, we first compute $X'(1),...,X'(n')$ where $X'(i,m) = \sum_{j=(i-1)d+1}^{(i-1)d+m} X(j)$ for $i = 1,...,n'$ and $m = 1,...,d$. Next we will simulate (as described below) the prefix circuit $\mathbf{C}_{n',b}$ with in-degree b, inputs $X'(1,d),...,X'(n',d)$ yielding outputs $Y'(1),...,Y'(n')$ where $Y'(i) = X'(1,d)\cdots X'(i,d)$ for $i = 1,...,n'$. Finally in time $O(d)$ using n' processors we compute $(X'(1,1),...,X'(1,d),Y'(1)\cdot X(2,1),...,Y'(1)\cdot X(2,d),...,Y'(n'-1)\cdot X'(n',1),...,Y'(n'-1)\cdot X'(n',d) = (X(1),X(1)\cdot X(2),...,X(1)\cdots X(n))$ as required.

The key problem remaining is to show that we can simulate $\mathbf{C}_{n',b}$ in time $O(d)$ using at most P processors. Note that each node v of $\mathbf{C}_{n',b}$ computes a binary number, say B_v, of at most $2\log n$ bits. We subdivide B_v into d blocks $B_{v,0},...,B_{v,d-1}$ each of s consecutive bits, so that $B_v = \sum_{i=0}^{d-1} 2^{si} B_{v,i}$. Our simulation of $\mathbf{C}_{n',b}$ will stream the computation of these blocks in d stages through the circuit, in order of the lowest precision to finally the highest precision (as in a carry-add circuit with base 2^s). The computation of $B_{v,i}$ in stage i depends on the carry value $F_{v,i}$ (which is 0 if $i = 0$, and else is the carry of size $< b$ from the computation of $B_{v,i-1}$ in stage $i-1$), plus the value of blocks $B_{u_1,i},...,B_{u_b,i}$ where $u_1,...,u_b$ are the nodes of $\mathbf{C}_{n',b}$ immediately preceding node v. We now show how to perform the computation of $B_{v,i}$ in $O(1)$ steps using $\leq P/n'$ processors. We will encode $F_{v,i},B_{u_1,i},...,B_{u_b,i}$ as an integer $I_{v,i} = F_{v,i} + 1 + b(\sum_{j=0}^{b-1} 2^{sj} B_{u_j,i}) \leq 2^{(s+1)b}$. We will precompute a table $f(1),...,f(2^{(s+1)b})$ such that $f(I_{v,i}) = B_{v,i}$. (Note that since this precomputation only involves summing $b+1$ integers of size $\leq O(\log n)$ bits each, it can be done in time $O(\log b)$ using a total of $b2^{(s+1)b} \leq 2^{(s+2)b}$ processors.) For each node v, we will associate $b+1$ processors $P_{a,0},...,P_{a,b}$ with each possible value $a \in \{1,...,2^{(s+1)b}\}$, and also a distinct variable $\text{VOTE}_{a,v,i}$ which is initially set to 1. The processors $P_{a,0},...,P_{a,b}$ will verify that $F_{v,i},B_{u_1,i},...,B_{u_b,i}$, respectively, encode $I_{v,i} = a$; if not and a conflict is detected by a processor $P_{a,j}$ then it sets $\text{VOTE}_{a,v,i}$ to 0. Hence there will be a unique value $a_o = I_{v,i}$ with $\text{VOTE}_{a_o,v,i}$ remaining set to 1 and for all other $a \neq a_o$, $\text{VOTE}_{a,v,i}$ set to O. We can let the distinguished processor $P_{a_o,0}$ set $B_{v,i}$ to $f(a) = f(I_{v,i})$. Thus we have computed $B_{v,i}$ in $O(1)$ time using $b2^{(s+1)b} \leq 2^{(s+2)b} \leq P/n'$ processors. Since $\mathbf{C}_{n',b}$ has depth $O(\log_b n) = O(d)$ and $O(n')$ nodes we require total time $O(d)$ and a number of processors upper bounded by $O(n')\cdot(P/n') = O(P)$. A constant factor slow down allows us to simulate $\mathbf{C}_{n',b}$ in time $O(d)$ using P processors. □

(Note that Theorem 5.1 implies that the parity of all prefixes of an n bit boolean sequence can be computed in time $O((\log n)/\text{loglog}(P\log n/n))$ using $P \geq n/\log n$ P-RAM processors. We leave as an open problem whether these upper bounds on processors and time are asymptotically optimal for the P-RAM model.)

Using similar speed up techniques, our optimal parallel integer sorting algorithm described in the proof of Theorem 3.1 is easily modified to achieve similar time and processor bounds.

COROLLARY 5.1. *n integer keys in $[n^{O(1)}]$ can be sorted in time $O(\log n/\text{loglog}(P \log n/n))$ using $P \geq n/\log n$ P-RAM processors.*

6. ACKNOWLEDGEMENTS

The author thanks S. Rajasekaran and Paul Spirakis for a careful reading of this paper.

7. REFERENCES

Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

Angluin, D. and L.G. Valiant, "Fast Probabilistic Algorithms for Hamiltonian Paths and Matchings," *J. Comp. Syst. Sci.* 18 (1979), pp. 155-193.

Ajtai, M., J. Komlós, and E. Szemerédi, "An $O(n \log n)$ Sorting Network," Proc. 15th Annual Symposium on the Theory of Computing, 1983, pp. 1-9.

Batcher, K., "Sorting Networks and Their Applications," Spring Joint Computer Conf. 32, AFIPS Press, Montvale, N.J., pp. 307-314.

Chernoff, H., "A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations," *Annals of Math. Statistics* 23 (1952).

Feller, W., *An Introduction to Probability Theory and Its Applications,* Vol. 1, Wiley, New York, 1950.

Fich, F.E., "Two Problems in Concrete Complexity: Cycle Detection and Parallel Prefix Computation," Ph.D. Thesis, Univ. of California, Berkeley, 1982.

Hoeffding, W., "On the Distribution of the Number of Successes in Independent Trials," *Ann. of Math. Stat.* 27 (1956), 713-721.

Hopcroft, J.E. and R.E. Tarjan, "Efficient Algorithms for Graph Manipulation," *Comm. ACM* 16 (6) (1973), 372-378.

Johnson, N.J. and S. Katz, *Discrete Distributions,* Houghton Mifflin Comp., Boston, MA, 1969.

Kucera, L., "Parallel Computation and Conflicts in Memory Access," *Information Processing Letters* 14 (2) (1982).

Leighton, T., "Tight Bounds on the Complexity of Parallel Sorting," 16th Symp. on Theory of Computing, Washington, D.C., 1984, pp. 71-80.

Ladner, R.E. and M.J. Fischer, "Parallel Prefix Computation," *J. Assoc. Computing Mech.* 27 (4) (1980), pp. 831-838.

Miller, G.L. and J.H. Reif, "Parallel Tree Contraction and Its Application," 18th IEEE Symposium on Foundations of Computer Science, Portland, Oregon, October, 1985.

Reif, J.H., "Symmetric Complementation," *J. of the ACM,* 31 (2) (1984a), pp. 401-421.

Reif, J.H., "On the Power of Probabilistic Choice in Synchronous Parallel Computations," *SIAM J. Computing,* 13 (1) (1984b), pp. 46-56.

Reif, J.H. and J.D. Tygar, "Efficient Parallel Pseudo-Random Number Generation," *CRYPTO'85,* Santa Barbara, CA, Aug. 1985.

Reif, J.H. and L.G. Valiant, "A Logarithmic Time Sort for Linear Size Networks," Proc. 15th Annual ACM Symp. on the Theory of Computing, (1983), pp. 10-16..

Reischuk, R., "A Fast Probabilistic Parallel Sorting Algorithm," Proc. of 22nd IEEE Symp. on Foundations of Computer Science (1981), pp. 212-219.

Shiloach, Y. and U. Vishkin, "Finding the Maximum Merging and Sorting in a Parallel Computation Model," *J. of Algorithms* 2 (1981), p. 88-102.

Tarjan, R.E., "Depth Forest Search and Linear Graph Algorithms," *SIAM J. of Computing* 1 (2) (1972), pp. 146-160.

Vishkin, U., "Randomized Speed-Ups in Parallel Computation," Proc. of the 16th Annual ACM Symp. on Theory of Computing, Washington, D.C., April 1984, pp. 230-239.

APPENDIX A: Probabilistic Bounds

The randomized algorithms in the preceding sections are analyzed by applying the following probabilistic bounds on the tails of binomial and hypergeometric distributions (see also [Feller, 80]).

Let random variable X *upper bound* random variable Y (and Y *lower bound* X) if for all x such that $0 \leq x \leq 1$, $\text{Prob}(X \leq x) \leq \text{Prob}(Y \leq x)$.

A.1 Binomial Distributions

A *binomial variable* X with parameters n,p is the sum of n independent Bernoulli trials, each chosen to be 1 with probability p and 0 with probability $1-p$. The *binomial distribution function* is $\text{Prob}(X \leq x) = \sum_{k=0}^{x} \binom{n}{k} p^n (1-p)^{n-k}$. The bounds of [Chernoff, 52] and [Angluin and Valiant, 79] imply

LEMMA A.1. *For all ϵ,p,n where $O \leq p \leq 1$ and $0 < \epsilon < 1$,*

$$\text{Prob}(X \leq \lfloor (1-\epsilon)pn \rfloor) \leq \exp(-\epsilon^2 np/2)$$

$$\text{Prob}(X \geq \lceil (1+\epsilon)np \rceil) \leq \exp(-\epsilon^2 np/3)$$

LEMMA A.2. **[Hoeffding, 56].** *Let $X_1,\ldots,X_n$ be independent binomial variables. Then $\sum_{i=1}^{n} X_i$ is upper bounded by a binomial variable with parameters n,p with mean $np = \sum_{i=1}^{n} \text{mean}(X_i)$.*

A.2 Hypergeometric Distributions

Fix p,s where $0 \leq p \leq 1$ and $0 \leq s \leq n$. Let A be a subset of $\{1,\ldots,n\}$ of size np. A *hypergeometric variable* Y with parameters s,np,n is defined as $Y = |S \cap A|$ where S is a random sample of s elements of $\{1,\ldots,n\}$ chosen without replacement.

Suppose we independently choose $s \leq n$ random integers $r_1,\ldots,r_s \in \{1,\ldots,n\}$. Let index i be *conflicted* if there exists distinct a,b such that $r_a = r_b = i$. Let Z be the total number of conflicted indices $i \in \{1,\ldots,n\}$.

LEMMA A.3. *Z is upper bounded by a hypergeometric variable with parameters s,s,n.*

[Johnson and Katz, 69] attribute the following bound to Uhlmann.

LEMMA A.4. *If X is binomial with parameters s,p and Y is hypergeometric with parameters s,np,n then*

$$\text{Prob}(X \leq x) > \text{Prob}(Y \leq x)$$

$$\text{for} \quad 0 < p \leq \frac{nx}{(s-1)(n+1)}$$

and

$$\text{Prob}(X \leq x) > \text{Prob}(Y \leq x)$$

$$\text{for} \quad \frac{(1+nx/(s-1))}{(n+1)} \leq p \leq 1$$